Software Design and Architecture

Lab # 09

Inheritance and Polymorphism in Java

Instructor: Fariba Laiq

Email: fariba.laiq@nu.edu.pk

Course Code: CL1002

Semester Spring 2023

Department of Computer Science,

National University of Computer and Emerging Sciences FAST Peshawar Campus

Inheritance in Java

1. Inheritance
2. Types of Inheritance
3. Why multiple inheritance is not possible in Java in case of class?

Inheritance in Java is a mechanism in which one object acquires all the properties and behaviors of a parent object. It is an important part of OOPs (Object Oriented programming system).
The idea behind inheritance in Java is that you can create new classes that are built upon existing classes. When you inherit from an existing class, you can reuse methods and fields of the parent class. Moreover, you can add new methods and fields in your current class also.
Inheritance represents the **IS-A relationship** which is also known as a *parent-child* relationship.

Why use inheritance in java

- For Method Overriding (so runtime polymorphism can be achieved).
- For Code Reusability.

Terms used in Inheritance

- **Sub Class/Child Class:** Subclass is a class which inherits the other class. It is also called a derived class, extended class, or child class.
- **Super Class/Parent Class:** Superclass is the class from where a subclass inherits the features. It is also called a base class or a parent class.
- **Reusability:** As the name specifies, reusability is a mechanism which facilitates you to reuse the fields and methods of the existing class when you create a new class. You can use the same fields and methods already defined in the previous class.

The syntax of Java Inheritance

```
1. class Subclass-name extends Superclass-name
2. {
3.   //methods and fields
4. }
```

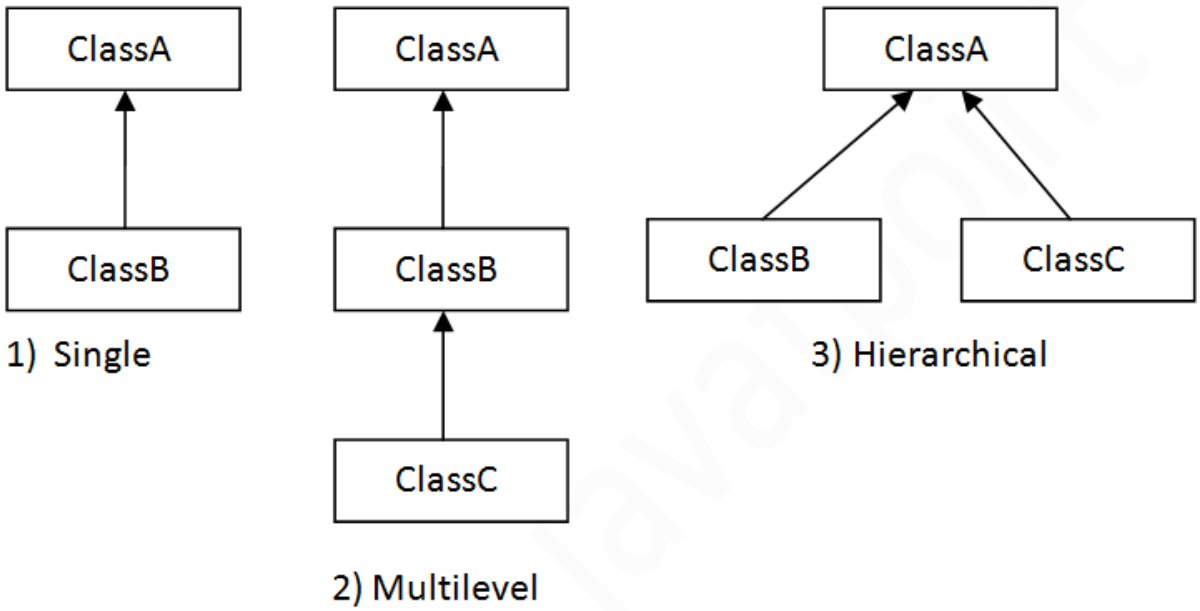
The **extends keyword** indicates that you are making a new class that derives from an existing class. The meaning of "extends" is to increase the functionality.
In the terminology of Java, a class which is inherited is called a parent or superclass, and the new class is called child or subclass.

Access Level for each Access Modifier

	default	private	protected	public
Same Class	Yes	Yes	Yes	Yes
Same package subclass	Yes	No	Yes	Yes
Same package non-subclass	Yes	No	Yes	Yes
Different package subclass	No	No	Yes	Yes
Different package non-subclass	No	No	No	Yes

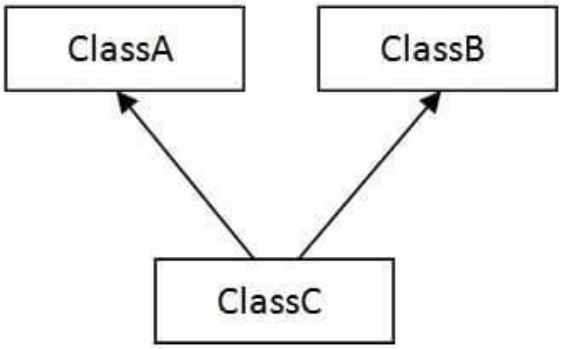
Types of inheritance in java

On the basis of class, there can be three types of inheritance in java: single, multilevel and hierarchical.
In java programming, multiple and hybrid inheritance is supported through interface only. We will learn about interfaces later.

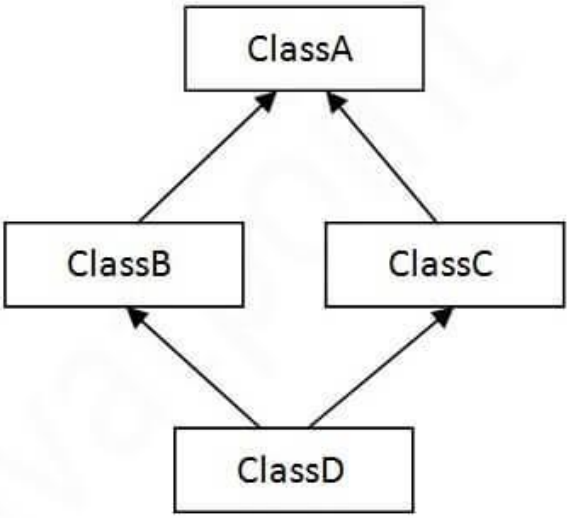


Note: Multiple inheritance is not supported in Java through class.

When one class inherits multiple classes, it is known as multiple inheritance. For Example:



4) Multiple



5) Hybrid

.

Java Inheritance Example

Below is the example of hierarchal inheritance in Java.
When two or more classes inherits a single class, it is known as *hierarchical inheritance*.

Hierarchal Inheritance

Below if the example of hierarchal inheritance. Note that the display () method is overridden in both sub classes. Note that if you declare a method final, you cannot override it in sub-class.

Person.java

```
public class Person {
    protected String name;
    Person(String name)
    {
        this.name=name;
    }
    public void display()
    {
        System.out.println("I am a person and my name is: "+name);
    }
    public String getName()
    {
        return this.name;
    }
}
```

Employee.java

```
public class Employee extends Person
{
   private int salary;
   Employee(String name, int salary) {
      super(name);
      this.salary=salary;
   }
   public void display()
   {
      System.out.println("I am a an employee. My name is: "+name+" and my
      salary is: "+salary);
   }
}
```

Student.java

```
public class Student extends Person {
   private int marks;
   Student(String name, int marks) {
```

```
        super(name);
        this.marks = marks;
    }
    public void display()
    {
        System.out.println("I am a student. My name is: "+name+" and my marks
        are: "+marks);
    }
    public int getMarks() {
        return marks;
    }
}
```

Main.java

```
public class Main{
    public static void main(String[] args) {
        Person p=new Person("Fariba");
        Student s=new Student("Ali", 90);
        Employee e=new Employee("Adil", 50000);
        p.display();
        s.display();
        e.display();
    }
}
```

Output:

I am a person and my name is: Fariba
I am a student. My name is: Ali and my marks are: 90
I am a an employee. My name is: Adil and my salary is: 50000

Multi-Level Inheritance Example

Person.java

```
public class Person {
    protected String name;
    Person(String name)
    {
        this.name=name;
    }
    public void display()
    {
```

```
        System.out.println("I am a person and my name is: "+name);
    }
    public String getName()
    {
        return this.name;
    }
}
```

Employee.java

```
public class Employee extends Person
{
    protected String designation;
    Employee(String name, String designation) {
        super(name);
        this.designation=designation;
    }
    public void display()
    {
        System.out.println("I am a an employee. My name is: "+name+" and my
        designation is: "+designation);
    }
}
```

HourlyEmployee.java

```
public class HourlyEmployee extends Employee{
    private int no_of_hours_worked;
    private int hourly_rate;

    HourlyEmployee(String name, String designation, int no_of_hours_worked,
    int hourly_rate) {
        super(name, designation);
        this.no_of_hours_worked=no_of_hours_worked;
        this.hourly_rate=hourly_rate;
    }
    public void display()
    {
        super.display();
        System.out.println("Hourly rate: "+hourly_rate+ " No of hours worked: "+no_of_hours_worked);
    }

}
```

Main.java

```
class Main
{
   public static void main(String args[])
   {
      HourlyEmployee e=new HourlyEmployee("Fariba", "Android app
      developer", 10, 5);
      e.display();
   }
}
```

Output:

I am a an employee. My name is: Fariba and my designation is: Android app developer

Hourly rate: 5 No of hours worked: 10

In the above program the inheritance has 3 levels. Person->Employee-> HourlyEmployee. Note that the display method of the Person class is overridden in the Employee class, which is further overridden in the hourly employee class. Also, the display method of the hourly employee class calls the display method of the Employee class using the super keyword, which prints the name, and the designation of the employee then it also prints the no of hours worked and the hourly rate variable of its own class. In Java, super is a keyword that refers to the superclass of a class. It is used to call the constructor, method, or variable of the superclass, or to differentiate between the superclass and subclass methods or variables with the same name.

Q) Why multiple inheritance is not supported in java?

To reduce the complexity and simplify the language, multiple inheritance is not supported in java.
Consider a scenario where A, B, and C are three classes. The C class inherits A and B classes. If A and B classes have the same method and you call it from child class object, there will be ambiguity to call the method of A or B class.
Since compile-time errors are better than runtime errors, Java renders compile-time error if you inherit 2 classes. So, whether you have same method or different, there will be compile time error.

Java Polymorphism

Polymorphism means "many forms". To put it simply, polymorphism in Java allows us to perform the same action in many different ways. There are two types of polymorphism in Java: compile-time polymorphism and runtime polymorphism. We can perform polymorphism in java by method overloading and method overriding.

Compile Time Polymorphism

In Java is also known as Static Polymorphism. Furthermore, the call to the method is resolved at compile-time. Compile-Time polymorphism is achieved through Method Overloading.

Runtime Polymorphism in Java

Runtime polymorphism or Dynamic Method Dispatch is a process in which a call to an overridden method is resolved at runtime rather than compile-time.

In this process, an overridden method is called through the reference variable of a superclass. The determination of the method to be called is based on the object being referred to by the reference variable. Also, it should be noted that runtime polymorphism can only be achieved through functions and not data members.

Overriding is done by using a reference variable of the superclass. The method to be called is determined based on the object which is being referred to by the reference variable. This is also known as **Upcasting**.

Upcasting takes place when the Parent class's reference variable refers to the object of the child class.

Shape.java

```
public class Shape {
    public void display()
    {
        System.out.println("I am a shape");
    }
}
```

Circle.java

```
public class Circle extends Shape{
    public void display()
    {
        System.out.println("I am a circle");
    }
}
```

Square.java

```
public class Square extends Shape{
    public void display()
    {
        System.out.println("I am a square");
    }
}
```

Main.java

```
class Main
{
    public static void main(String args[])
    {
       Shape s=new Circle();
       s.display();
    }
}
```

Output:

I am a circle

Explanation:

The Main class creates an object of the Circle class and assigns it to a reference variable of the Shape class type. When the display() method is called on the s object, it calls the display() method of the Circle class, not the display() method of the Shape class, because the Circle class has overridden the display() method. This is an example of polymorphism in Java, where an object can take on different forms depending on the reference variable used to refer to it. In this case, the Circle object is treated as if it were a Shape object, allowing the display() method of the Circle class to be called through the Shape reference variable.

In the Main class, the Circle class is used to create an object, and that object is

assigned to a reference variable of type Shape. However, the actual object that is created is still a Circle object, which means that the display() method of the Circle class will be called when the display() method is invoked on the s object.

This is an example of dynamic method dispatch, where the method to be called is determined at runtime based on the actual object that is being referred to, rather than the reference variable type. Because the Circle class overrides the display() method of the Shape class, the display() method of the Circle class is called when the display() method is invoked on the s object, even though the reference variable is of type Shape.

Note: What would happen if we add a method foo() in Circle class and call it with the reference variable of the Shape class? Try and check it out.

Compile-Time Polymorphism vs. Run-Time Polymorphism

Compile-Time Polymorphism	Run-Time Polymorphism
• The method call is handled by the compiler	• The compiler cannot control the method call in run-time
• Compile-Time Polymorphism is less flexible, as it needs to handle all method calls in compile-time	• Run-Time Polymorphism exhibits higher flexibility as the method calls get handled at run-time
• Integrating the right method call with the proper method is done in compile-time	• Combining the correct method call with the right method is done in run-time
• Occurs during Method Overloading and Operator Overloading	• Occurs during Method Overriding

Static Binding and Dynamic Binding

Connecting a method call to the method body is known as binding.
There are two types of binding

1. Static Binding (also known as Early Binding).
2. Dynamic Binding (also known as Late Binding).